# L'instruction conditionnelle `switch`

J. Sam, J.-C. Chappelier, V. Lepetit

version 1.0 de octobre 2013

En C++, on peut écrire de façon plus synthétique l'enchaînement de plusieurs conditions dans le cas où l'on teste différentes valeurs d'une expression retournant un entier .

## 1 L'instruction `switch`

Ainsi une séquence d'instructions `if` telle que celle-ci :

```
if (i == 1)
  Instructions 1
else if (i == 12)
  Instructions 2
  ...
else if (i == 36)
  Instructions N
else
  Instructions N+1
```

peut avantageusement être remplacée par la tournure suivante :

```
switch (i) {
case 1:
  Instructions 1;
  break;
case 12:
  Instructions 2;
  break;
...
case 36:
  Instructions N;
  break;
default:
  Instructions N+1;
  break;
}
```

Lors de l'exécution d'un instruction `switch`, l'expression suivant le mot clé `switch` est évaluée [1]. L'exécution se poursuit alors au niveau du `case` correspondant au résultat de l'évaluation.

Cette exécution continue en séquence, sans aucune évaluation de condition, jusqu'à ce que le mot réservé `break` soit rencontré ou que la fin de l'instruction soit atteinte.

La première portion de code ci-dessus n'est donc équivalente, du point de vue de l'exécution, qu'à un `switch` où chaque cas de termine par un `break` (comme c'est le cas de l'exemple ci-dessus).

Dans le cas où l'évaluation de l'expression ne retourne pas une valeur correspondant à un `case`, le programme exécute les instructions correspondant au cas par défaut (précédées du mot clé `default`).

## 2 Rôle du `break` : exemple

Examinons l'exemple suivant :

```
switch (a+b) {
  case 2:
  case 8: instruction2; // lorsque (a+b) vaut 2 ou 8
  case 4:
  case 3: instruction3; // lorsque (a+b) vaut 2, 3, 4 ou 8
    break;
  case 0: instruction1; // exécuté uniquement lorsque
    break;              // (a+b) vaut 0
  default: instruction4; // dans tous les autres cas
    break;
}
```

Supposons que l'évaluation de l'expression `(a + b)` retourne `8`. L'exécution va se dérouler comme suit :

1. le programme va se « brancher » au cas `8` et ainsi exécuter l'instruction `instruction2` ;
2. comme il n'y a pas d'instruction `break` en dessous, l'exécution se poursuit et traverse le cas 4 (sans faire de test et sans rien faire puisqu'il n'y a pas d'instructions à exécuter) ;
3. c'est ensuite `instruction3` qui est rencontrée et exécutée ;
4. on rencontre finalement le `break` en dessous de `instruction3` et l'on peut ainsi sortir de l'instruction `switch`.

On voit donc ainsi que
– on exécutera l'instruction `instruction2` si `(a + b)` vaut 2 ou 8 ;
– on exécutera l'instruction `instruction3` si `(a + b)` vaut 2, 3, 4 ou 8 ;
– on n'exécutera `instruction1` que si `(a + b)` vaut 0.

1. dans l'exemple qui précède, l'expression se réduit à la simple variable `i`

Le fait de ne pas mettre d'instruction en face d'un cas permet simplement de mettre en œuvre un « *ou logique* » entre plusieurs conditions : on exécute `instruction2` si `(a + b)` vaut 2 **ou** 8 par exemple. Cela permet d'éviter des répétitions inutiles de code. Mais il vaut peut être mieux dans un premier temps éviter ce genre d'optimisation, peu facilement compréhensible à la première lecture...

# 3 `switch` vs `if..else`

`switch` est moins général que `if..else` :

– la valeur sur laquelle on teste doit être soit de type intégral : `char`, `int`, `unsigned int` etc.)
– l'expression testée est toujours la même (le « `i` » ou « `a+b` » des exemples précédents) ;
– les cas doivent être des constantes (pas de variables).